THIS DOC IS PUBLIC

Universal Context Management RFC

Author: Stephen Belanger
Last Updated: May 6, 2024

Changelog

- Mar 27, 2024 : Initial version
- Apr 10, 2024 : Added Calling Context interfaces
- Apr 13, 2024 : Built out `ContextVariable` and `ContextFrame` types to be fully-functional and include multi-store runs to integrate better with Diagnostics Channel. Also marked as in-review.
- Apr 30, 2024 : Further explained separation of calling and continuation contexts, and added explanation of coroutines being internally synchronized
- May 2, 2024 : Split up some sections to put problem-related text in the Problem Statement section

Problem Statement

Building structural insight data about an application without changing application behavior requires being able to propagate data alongside application flow without explicitly passing it through user code. We call this context management. Conceptually how it needs to work is the same across all languages, it needs to flow in the same direction that application data and execution flows.

How this is implemented across languages is quite different and many don't include the capability at all. We often need to do it ourselves externally which may require significant modifications to the runtime or language environment to allow us to pass this data through the execution graph externally.

We have created different solutions for different languages and in many cases provide only partial solutions with some less common cases left unresolved, for example most languages only propagate into calls but not back out, and some languages don't propagate context over threads when work transitions between threads.

Globals but synchronized to execution

In a language completely void of asynchrony, context management can simply be stored in globals as there is no point at which execution will transition to something else which would invalidate that global state. When any level of async scheduling is added it becomes necessary to capture and restore context around any barriers between where a continuation is queued and where the resolution then executes it.

Context forms a graph of changes which follows the logical execution flow of the system. If the flow of the system is non-linear then it becomes necessary to reroute the edges of the context graph to follow the same paths of data and execution flow within the application.

Layered rearrangement

Asynchrony is generally not a single-tiered system. It's useful to think of asynchrony as layers of execution rearrangement moving outward from the runtime core, through higher-level constructs in libraries, and often into user code. Languages in which asynchrony exists tend to treat functions or closures as values which can be stored and run by code at any time, so additional layers of execution rearrangement can occur in libraries or even in user code.

For example, the JavaScript runtime has microtasks rearranging execution and those would link back to the point at which they are scheduled. A connection pooling library introduces an additional layer of execution rearrangement around when connections are acquired and released. App code could then be using async/await and especially `Promise.all(...)` in ways which rearrange execution of the application in interesting and *user-facing* ways. Flexibility is required to follow a context path which makes sense through all these layers.

To flow data correctly through an application with many layers of execution rearrangement it is essential to have tools to be able to influence how context data flows at each level of execution. It's likely pattern-specific tooling would be required around some of the lower-level systems like coroutine syntax such as fibers, generators, or async/await.

Unidirectional context systems

Some languages already have context systems of some variety which provide *part* of the solution for what observability products need. The typical design is that context systems flow values into calls and sometimes into async scheduled tasks. However it's *not* typical for them to flow data *out* of calls. This can be thought of as implicit input arguments and implicit return values. Existing systems primarily cover the implicit input argument case but rarely cover the case for implicit return values.

The *need* for the return path is essential when it comes to syntax-assisted async such as async/await, generators, or fibers as OpenTelemetry requires that the next task after execution merges *back* needs to include follows-from links to the task which occurred within the prior call. At present it is impossible in most languages to follow the OpenTelemetry spec for providing follows-from links with auto-instrumentation.

Branching and merging

Execution in an async application branches and merges at various points defined by the rearrangement of execution described above. The path in which context flows *always* needs to flow outward into calls, and this is what existing unidirectional context systems achieve, however it is also often needed to propagate back when branches of execution converge. These merge paths, which also generally correspond to returns, are more complex as there can often be cases where multiple branches are merging back and so it's not a clear task of just propagating whatever value is held.

Sometimes it requires additional logic on the part of the reader to decide from which path to read. But also, because context flow decisions need to be made continuously, as the execution graph resolves, those read decisions also often need to occur at the right time for the appropriate decision to be made.

If you think of the execution of the application code as a graph, there are two paths worth considering, representing the user code and the internals *behind* calls made in user code. Following the internal path includes more detail, but that detail can obscure user intent in some cases such as with connection pools where acquiring a connection may appear as a continuation of the code which *released* the connection rather than the code which *requested* the connection.

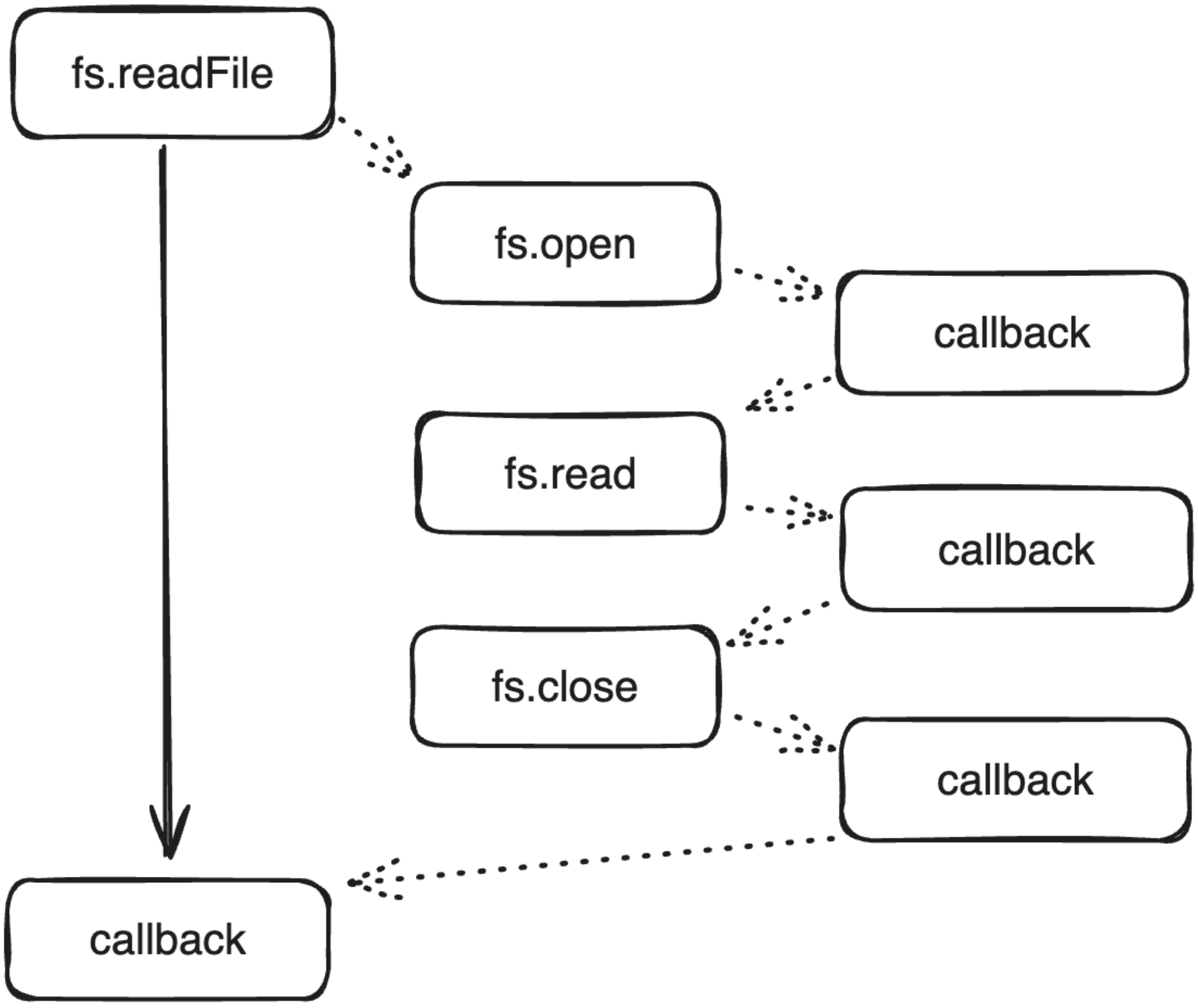
In cases such as these it might make sense to always follow the user code path rather than the internal path, and indeed it is *suggested* that the ideal path is to follow the most user-facing code path. However, it's *also* worth considering that while it sometimes makes sense to reduce the graph to the user code path, it can be difficult without proper tools to return to the internal graph when the user code path has already been selected. Additionally, it can be a bit unclear where user code ends and internals begin with things like user-initiated asynchrony such as promise branching and merging tools as `Promise.all(...)` in Node.js.

When the user code path is followed it will orphan any activity internal to the calls made from that user code. However the branch *could* flow the context normally up *until* the point where the other context would be restored, replacing what had propagated automatically through that branch. This *does* mean that with the right tools the original context before the change can be captured and restored or read from after the transition occurs.

Multiple paths problem

When dealing with graph branching and merging there's often two paths which could be taken for propagation, a direct path and an internal path. Selecting the correct path depends on what level of granularity we want in the insight we gather.

The following example represents the graph of calling `fs.readFile(path, callback)`. Internally it makes several lower-level filesystem calls, represented by the dotted line, which may be relevant to capture. But by following the user path, represented by the solid line, it will orphan the sub-graph of the internal execution.



There are two different points where path decision *could* apply: if applying *before* entering the internal code then that code would be run with an empty context, whereas if applying the path when the callback is reached the internal graph can actually *continue* from the context at the point the call begins and simply restore to the outer state at the point where the callback begins.

Design

Architectural Overview

Context needs to be stored somewhere separate from the user code to allow systems to access the data in the future when it is needed. A user-friendly way to achieve this is with a container class where instance references can be used from anywhere as an indirect way to acquire an associated value. To allow different products to store different things and for different pieces of context data to change at different times in an efficient way it needs multi-tenancy and tools to control the context flow both globally and per-store.

The model being proposed is to have a container class which represents a specific value with interfaces to get the current value from the container and to adjust the current value. Changing of values would be split into two forms:

- Synchronous call windows which form a stack of current values and pop back
- Async continuations which link back to a snapshot taken at the point the initial work was scheduled

Windows

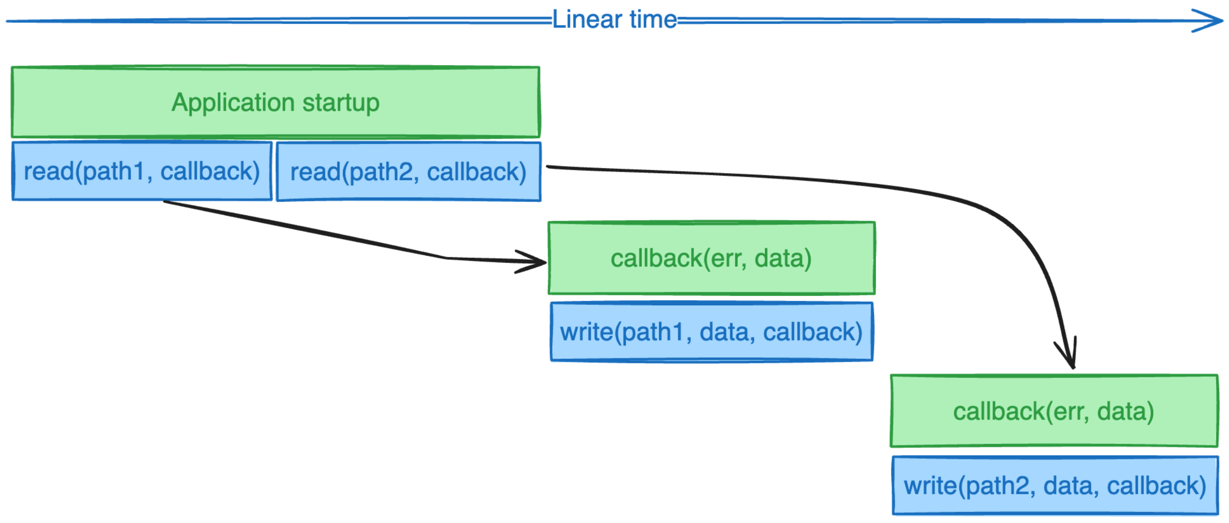
A window is a block of synchronous execution which is in some way delimited to mark the start and end of that execution. The context manager can be given a value to propagate into any nested calls run within that window. At this level it is essentially like storing the value globally to be read in nested code from the global. It can also be interpreted as implicit call arguments passed into any nested call until the value changes.

Asynchronous code would have a scheduling point within a call window where it would capture the current context value and store it to be restored when another window around the continuation of that async task is executed.

The exact implementation of a window is left undefined; it could be any form of scoping mechanism, but should conform to the concept of being a synchronous sequence of logically grouped execution. In the case of Node.js this is implemented as the boundary of a function call. In other languages there may be other options such as RAII scopes. Each language should make their own clear definition of what their relevant windows of synchronous execution are.

Some windows would be determined automatically by the runtime, such as a promise continuation having a window around the continuation function which restores the value propagated from the point where the task was scheduled. Additionally, new windows could be user-provided on a per-store basis either through the interfaces described below or through the use of those interfaces via integration with Universal Diagnostics Channel RFC.

In the following diagram, blue boxes represent call windows and green boxes represent continuation windows. The entry of the application is considered a continuation of the act of starting the application. This is a continuation and not a call window because there is no way at that point for a context to already be present unless provided external to the runtime, in which case it should be considered a continuation of the source of the application starting. For example, a parent process may start a sub-process and the entry of that sub-process would be considered a continuation of the parent process.



Call Window

Synchronous calls form a stack, and so context for sync calls also needs to form a stack. It's useful to differentiate sync calls from continuations as async continuations have no immediate stack of their own but it may make sense for them to restore the stack as it was when the continuation was scheduled.

There's an additional differentiation between sync calls and async continuations though, if a sync call produces a span and then another sync call within it produces another span it should have an OpenTelemetry child-of relationship to the parent span. Whereas a sync call within a continuation should have a follows-from relationship.

Continuation Window

A continuation window occurs when an async task is scheduled and it returns to the application in some way to execute, possibly consuming a value of some sort. Typical examples are callbacks or promise continuations.

When a continuation occurs it should restore the context value captured at the point where it was scheduled, but with a slight difference that if a sync call occurs within the continuation it should have a OpenTelemetry follows-from relationship to the parent stored in the context rather than a child-of relationship. For this reason it is useful to separate the concepts of call windows and continuation windows with at least some sort of marker indicating that the context value did not originate directly from the currently executing code.

Propagation through asynchrony

Synchronous windows are fairly straightforward as the stored value simply remains the same until the end of the window where it restores to the value it had before the window began, treating it like a stack. Where things get a little more complicated is with asynchronous programming.

All async programming has some discrete indicator of the completion of execution of an async task, though that indicator may be deeper than is directly expressed. Typically this is a callback or a value container such as a Promise or Future. In some cases there are more complicated or repeatable windows of execution such as with Go channels where a send results in a continuation window of execution related to that send between where it completes the receive to where it begins the next receive, or the thread or goroutine using the channel ends. Some more advanced forms like Mutex or other low-level locks may be more challenging to express.

Another source of asynchrony which is a common point of confusion are patterns layered over other async things. For example, Node.js has event emitters–they are not async themselves but *become* async through use by *something else* that is async. It is important to differentiate between things which are *themselves* async and things which *inherit* asynchrony when defining where context should be propagated.

Generally speaking the boundaries we want to link automatically are around things which are *themselves* async. Linking other things can result in orphaned branches of execution and therefore orphaned sets of spans. This scenario *is* recoverable with the right tooling, so the *default* path taken is not too important, but for usability in typical cases this is the path we go for. Window Channel can also be used to *suggest* a context change rather than *enforcing* it.

Selecting optimal execution graph paths

It is important when execution branches and then converges that the most appropriate context value is selected at the point of convergence. Which path is most relevant could vary on a case-by-case basis, and in some cases *all* branches could have relevant data. For example, OpenTelemetry specifies that a span produced after a graph convergence should have follows-from links to *all* the contexts which were part of that convergence. This is particularly relevant in situations such as a `Promise.all(...)` converging any number of concurrent promise branches back to one point.

Making the decision at the continuation point is what Node.js does and I suggest this is the more suitable place to apply that decision as we can make inner spans linked to the user code and add to the trace before escaping their context back to the user code context. We can also make the internal path accessible *beyond* the restore point by capturing the calling context before the restore, which we already need to do anyway to restore after *completing* the context-bound callback.

Calling versus continuation context

Most context systems conflate context state between where it is set and captured to propagate into continuations with where it restores context around continuations but it’s actually helpful to consider these windows separately.

Let’s consider OpenTelemetry for a moment–when running a sync call within another sync call it is considered a child-of relationship while running a call in a callback or other async continuation is considered a follows-from relationship.

If you create a span and store it in the context and then within the sync execution of the window that context initially applies, another span is created, that span should be considered a child of the one currently stored. If a span is created directly in a continuation window from a task scheduled within that original call window then it should be considered a follows-from relationship to the span stored in the original call window.

It is therefore useful to store the context data in two slightly different ways between the two windows of execution. The suggested way is to differentiate internally between “current” context and “inherited” context. Anywhere that a “current” context is unavailable, retrieving the value could fallback to the “inherited” context, but it should be possible to retrieve the “current” context explicitly without fallback to be able to identify when the window setting the context value is actively running.

When propagating, the “current” context would be checked first and then, if absent, the “inherited” context would be used. You can also think of “current” context as being a stack *on top* of the “inherited” context at the root of whichever execution window the application is in at the time. At the top level of the application there would be an initial root context with no value. In this way a context stack can never be “empty” per-se, only that its root at any point may not contain a value, but still expresses a clear ownership through the execution graph of the application.

Coroutines are internally synchronized

In general cases, child-of describes a sync call within another sync call, but it can be more helpful to think of it as deterministically flowing execution within the parent in which it is called. This corresponds to a syntax which is growing in popularity: async/await.

Each await is a child-of the “current” context when the async function began, while each await after the first is a follows-from of any context created within the previous. The reason for this is that each of the awaits within the function are essentially synchronous from the perspective of the function itself executing. An async function executes in a deterministic order.

By constructing graphs of both child-of relationships *and* follows-from relationships it is possible to analyze the code in both internally linear and global, non-deterministic arrangements while presenting linear arrangements in a friendlier, more sync-like manner.

```javascript
async function process() {
  while (true) {
    // Getting a task is a child-of process(). It's also likely receiving
distributed
    // tracing headers with the task to restore a context around its
continuation of
    // which the await completion is the logical beginning.
    const task = await getTask()

    // Processing of the task here is both a follows-from to getting the task
but also
    // is a child-of process() as it is, from the perspective of this function
code,
    // effectively a determinstic flow. If the exact same code was written
synchronously
    // with await keywords removed it would be considered child-of so it should
also be
    // considered as child-of here.
    await processTask(task)
  }
}
```

In the example above, if `getTask()` internally stores a new span in the context, it needs to be possible for it to flow out of the await merging execution back into `process()` and into the call to `processTask(task)` later. To retrieve this follows-from linkage we make use of the "inherited" context while otherwise we would use normal context value to retrieve child-of relationships.

Interfaces & APIs

Exact naming of these interfaces are less important than the patterns presented, but it's all built around the container class concept.

ContextVariable

A `ContextVariable` from the user perspective behaves like a container which holds a value. That held value will change over time as provided windows and propagations swap out that internal value.

```JavaScript
// The ContextVariable type is a container for a context value. The value it
represents
```

```
// will change over time as the application switches between executing
different parts
// of the code. Within a request it could be used to change the value of the
current span
// Within the broader process another store could represent the current root
span.
// From user perspective it would seem that each store contains its own data
and
// propagates independently, however the ContextVariable is actually only a map
key into
// a non-multi-tenant ContextFrame map type.
//
// Node.js calls this AsyncLocalStorage, the name is less important than the
pattern.
class ContextVariable {
  // Retreive the current value from the context variable container. As
ContextVariable
  // is just a map key into a ContextFrame, what it actually does it get the
current
  // ContextFrame and get the value from that. If current frame is not set, get
from
  // the inherited context frame.
  get() {
    if (ContextFrame.current.has(this)) {
      return this.getFromActiveContext()
    }
    return this.getFromInheritedContext()
  }

  // Attempt to get a context value only from the active context. This will
only get
  // the value while the call window is active executing. This can be acquired
to
  // create child-of relationships.
  getFromActiveContext() {
    return ContextFrame.current.get(this)
  }

  // In some cases a static bind may have been applied which a specific store
may want
  // to break out of to get the context data from the internals path. To do so,
the
  // ContextFrame at the point a bind is run is captured allowing values from
it to
```

```
  // be accessed. This can be acquired to create follows-from relationships.
  getFromInheritedContext() {
    return ContextFrame.inheritedContext.get(this)
  }

  // Static bind captures the state of ALL context variables and returns a
function
  // which restores all those values before running the given function.
  // The function should retain the same signature and behaviour.
  //
  // In most cases static bind should be used, for example with connection
pools.
  // In rare cases, a specific variable may want to skip over some internal
flows,
  // so instance-scoped bind is available for those cases.
  static bind(fn) {
    return ContextFrame.bind(fn)
  }

  // Instance bind to restore the current state of this context variable when
the
  // function is run. This is sometimes useful when the true calling path
includes
  // internals which may not be relevant. Generally static bind is more
suitable.
  // Instance bind is mainly for edge-cases where we want a simpler graph.
  bind(fn) {
    const value = this.get()
    return function wrapped(...args) {
      return this.run(value, fn.bind(this, ...args))
    }
  }

  // Set the value of the context variable for the provided window. This value
will
  // propagate into any nested execution, either sync or async, until run(...)
is
  // called again on this store within that nested execution or until the end
of
  // all branches of nested execution have been reached.
  run(value, window, ...args) {
    return ContextFrame.run(this, value, window, ...args)
  }
}
```

```
// Here a variable is created and then its value is set for a window with run()
const variable = new ContextVariable()
variable.run('foo', () => {
  const value = variable.get() // 'foo'
})
```

Some languages like C++ may have RAII concepts which can be used for more language-intuitive scoping.

```
C/C++
class ContextVariableScope;

template <typename Type>
class ContextVariable {
  friend class ContextVariableScope;
  public:

  // Get the current value stored in the context variable
  Type get();

  // Run the given scope function with the current value of the store set to
the given
  // value for the duration of the scope function execution. After the function
completes
  // it will restore the previous state of the context variable and return the
value
  // returned by the scope function.
  //
  // We may not want the run(...) function at all in languages with more
suitable
  // concepts to express the scope.
  template <typename Return>
  Return run(Type value, std::function<Return()> scope);

  protected:
  // Store a new value in the context variable and return the previous value.
  Type exchange(Type value);
}

// Sets a context variable to a given value for a RAII scope
template <typename Type>
```

```
class ContextVariableScope {
  public:
  // When the scope constructs it exchanges the context variable value
  ContextVariableScope(ContextVariable<Type>& variable, Type value) {
    previous = variable.exchange(value);
  }

  // When the scope destructs it restores the prior context variable value
  ~ContextVariableScope() {
    variable.exchange(previous);
  }

  private:
  Type previous;
}

// Create a context variable container
auto variable = new ContextVariable();
{
  // Set the context variable value until the end of the block
  ContextVariableScope scope(variable, "foo");

  variable.get() // "foo"
}
```

Other languages may have other more appropriate windowing concepts. The point here is not to prescribe a specific way to implement windows but only to define the requirement that there is a distinct window of sync execution for which a value can be provided to use as the propagation value which will be received by any nested execution, both synchronous and asynchronous continuations of tasks scheduled within that window.

For any languages which opt for a function as a scoping mechanism, it is likely also desirable for the `run` function to pass arguments through to the given window function allowing for any general function call to be wrapped in a context window without introducing additional closures. Node.js does this presently with `AsyncLocalStorage`, so a call like `func(1, 2, 3)` can become `store.run(context, func, 1, 2, 3)`.

ContextFrame

A `ContextFrame` is the actual storage mechanism behind `ContextVariable` instances and behaves a bit different from how the user-facing surface area of `ContextVariable` may suggest. Rather than each `ContextVariable` containing its associated value, it actually only functions as a map key into the current `ContextFrame`. The `ContextVariable` type is multi-tenant while only a single `ContextFrame` will be active at any given time.

By making the `ContextVariable` a map key into a `ContextFrame` the most common interaction of propagating all the stores becomes a very low-cost reference swap. The other interactions of changing and reading the store become slightly more expensive due to the less direct interaction, but the trade-off is generally worth it when propagation happens significantly more frequently.

A `ContextFrame` is also immutable. Any time a store would change a new `ContextFrame` is created which copies the values from the current frame, setting the changed values only in the constructor for the new frame. This is what allows context propagation to be nothing more than a pointer copy to the current frame.

Note that the `ContextFrame` internal architecture is meant to be an optimization suggestion. Some languages which need to propagate over async barriers a lot less frequently may prefer more direct storage of values in `ContextVariable` instances. However, I believe in *most* cases that the frame-based approach has the best balance of performance and flexibility.

```javascript
// A context frame is a map of stores to their corresponding values. A map is immutable,
// replacing itself any time it would be modified.
class ContextFrame extends Map {
  // There is always only a single context frame active
  static current: ContextFrame

  // Keep track of what context we were in before we transitioned to
  // this one. This allows breaking out of context binds, if necessary.
  // Stores a blank frame by default to represent the application root.
  static inheritedContext: ContextFrame = new ContextFrame()

  // A new ContextFrame is created ONLY when a ContextVariable would change.
  // This allows the most common case of propagation of context data to only
  // need to copy a single ContextFrame reference rather than needing to deal
  // with the multi-tenancy or copy the state of all stores on every propagation.
  static run(store, value, window) {
    return this.runAll([[store, value]], window)
  }

  // Window Channel integration may need to update many bound stores at once.
  // Rather than producing a new frame for each change we can bulk-apply
changes
  // with only one new ContextFrame created.
  static runAll(pairs, window) {
    const frame = new ContextFrame(pairs)
    return frame.run(window)
  }

  // Captures the current frame and wraps the given function to re-enter the
frame
  // whenever the function is called. This is how propagation happens
internally.
  // Likely languages/runtimes will have their own internal way to do this more
  // optimally, but this is essentially all propagation needs to be--storing
the
  // pointer to the frame and applying it later, which is what run(...) does
  // around a function.
  static bind(fn) {
    const frame = ContextFrame.current
    return (...args) => frame.inherit(fn, ...args)
  }

  // When a new ContextFrame is constructed it copies all entries from the
current
  // ContextFrame into the new frame, then it sets any values which have
changed.
  constructor(changes) {
    super(ContextFrame.current)
    for (let [store, value] of changes) {
      this.set(store, value)
    }
  }

  // When the context frame runs, it swaps itself in as the current frame and
then
  // restores the prior frame after running the scope. This produces the
synchronous
  // call window stack behaviour.
  run(window, ...args) {
    // Swap to this frame as the active frame.
    const priorContext = ContextFrame.current
    ContextFrame.current = this
    try {
      return window(...args)
    } finally {
      // After the given scope function ends, restore the prior context frame.
      ContextFrame.current = priorContext
    }
  }

  // When a continuation window runs the context frame should be inherited.
  // This sets this context frame as the inherited frame to differentiate from
  // actively running context windows so child-of and follows-from
relationships
  // can be identified. It functions something like a stack root so at no point
  // should the context ever have no frame at all.
  inherit(window, ...args) {
    // Set this frame as the inherited frame.
    ContextFrame.inheritedContext = this
    try {
      return window(...args)
    } finally {
      // After the given scope function ends, clear inherited context frame.
      ContextFrame.inheritedContext = undefined
    }
  }
}
```

Similar to `ContextVariable`, some languages will likely prefer different patterns from function scopes for managing the current `ContextFrame` value. For example, a RAII scope for C++ might look like this:

```
C/C++
// Activate a context frame for a RAII scope
template <typename Type>
class ContextFrameScope {
  public:
  // When the scope constructs it exchanges the context frame
  ContextFrameScope(ContextFrame<Type>* frame) {
    previous = ContextFrame.current;
    ContextFrame.current = frame;
  }

  // When the scope destructs it restores the prior context frame
  ~ContextFrameScope() {
    ContextFrame.current = previous;
  }

  private:
  ContextFrame<Type>* previous;
}

// Create a context frame for a context variable change.
//
// Any time the user or runtime wants to capture the current state of every
context
// variable they can simply store ContextFrame.current and restore it later
with
// ContextFrameScope.
auto variable = new ContextVariable();
auto frame = new ContextFrame(variable, "foo");

callAfterTimeout([frame]() {
  // Set the context frame until the end of the block
  ContextFrameScope scope(frame);

  variable.get() // "foo"
}, 1000);
```

Deployment

Ideally this context pattern should be implemented as part of the languages or runtimes. We will likely need some languages to maintain it separately due to the time and challenge it will take to contribute upstream and get versions with the feature adopted by users. We should aim long-term at language and runtime adoption eventually though as implementation at that level can have much tighter integration and therefore much better performance potential.

Scalability

The key points which need business logic are: when a variable is changed, when a variable is extracted, and when a value should propagate. In most cases propagation will happen many, many times more frequently (1000x+) than extraction which is itself typically more than changing.

If the storage model follows the `AsyncContextFrame` pattern then the most frequent action of propagation should have very little cost as it only needs a reference copy. Extraction is also fairly low-cost as it's only a map read.

Only changing becomes more costly due to needing to do the map clone, but any alternative which allows multi-tenancy also requires a map clone at one of these levels.

Reliability

With separate store instances which can run and bind scopes separately the flow can follow a consistent default and additional paths through the execution graph can be drawn on a case-by-case basis. This allows enough control that any issues in context propagation can generally be worked around.

Security

By using a `ContextVariable` instance as a map key into a `ContextFrame`, associated data is only accessible if the store is accessible, so access can be secured effectively by limiting access to the `ContextVariable` instance. As long as `ContextFrame` is not exposed, the data it contains can not be retrieved. This prevents different projects from accidentally using or altering each other's data and causing data leaks.

Testing

Testing is fairly straightforward for the most part. Changing of the `ContextVariable` value needs checks before and after each transition to verify the correct change occurred and at the correct timing. There would be some more complex tests required around any automatic propagation flows we have the languages or runtimes make. For example, Node.js propagates into the callback or Promise continuation of any async task. This requires some more complicated async programming, but most propagation testing should be straightforward.

Key Milestones

The first step would be just providing the `ContextVariable` types in isolation. From there we can begin integrating that with any needed automatic propagation points.

Success Metrics

The core purpose of this RFC is to unify on a pattern and hopefully naming conventions, as much as possible. So the success criteria would be for a developer to be able to work with any of our supported languages and be able to propagate their context data without needing to learn new or significantly different mechanisms for doing this.

Open Questions

The proposal as it is presently is predicated on the idea that there is a specifically scoped window of execution for which we want propagation to occur. This may not be the case with some of the more exotic patterns used in some languages. We could alternatively consider only the transition point *into* a context if we ensure that all execution in the language will *always* adjust the scope so a transition can't leak into unrelated execution. For example transitioning to a new context within a function call might persist after the function completes and some *other* execution begins.

Alternative options

AsyncLocalStorage

Node.js at present includes an interface called AsyncLocalStorage which *mostly* matches the behavior described here, with a few omissions I would like to deal with. As it is constructed presently, it depends on some relatively expensive internal APIs and lacks much of the structural optimization described here along with some features such as calling context retrieval to build follows-from graphs and differentiation between current and inherited context.

AsyncContext

TC39 is preparing a proposal for context management in the JavaScript language. The present design has many open questions remaining about what is the most "correct" path to take as the current design does not allow supporting multiple paths. It also lacks the ability to control propagation at a per-store level.